Adaptive Optimizing Compilers for the 21^{st} Century

Keith D. Cooper *Devika Subramanian* *Linda Torczon*
Department of Computer Science
Rice University
6100 Main Street, MS 132
Houston, TX 77005

Abstract

Historically, compilers have operated by applying a fixed set of optimizations in a predetermined order. We call such an ordered list of optimizations a compilation sequence. This paper describes a prototype system that uses biased random search to discover a program-specific compilation sequence that minimizes an explicit, external objective function. The result is a compiler framework that adapts its behavior to the application being compiled, to the pool of available transformations, to the objective function, and to the target machine.

This paper describes experiments that attempt to characterize the space that the adaptive compiler must search. The preliminary results suggest that optimal solutions are rare and that local minima are frequent. If this holds true, biased random searches, such as a genetic algorithm, should find good solutions more quickly than simpler strategies, such as hill climbing.

1 The Changing Landscape

Each year, microprocessors, microcontrollers, and other computing engines find application in new areas. The number of computers in use is growing rapidly, as is the diversity of those systems. The variety among processors—different instruction sets, different performance parameters, different memory hierarchies—is increasing. Even commodity microprocessors come in low-power versions, embedded versions, and a variety of high-performance versions; the performance parameters of these versions can differ widely. Processors have also grown more complex, with multiple functional units, exposed pipelines, and myriad latencies that must be managed. In most cases, these computers execute code produced by a compiler—a translator that consumes source code and produces equivalent code for some target machine.

At the same time, the application of computing to new problems has created demand for compilers that optimize programs for new criteria, or new *objective functions*. In the 1980s and the early 1990s, the speed of compiled code was the dominant concern of users. In the late 1990s, the size of compiled code became an issue, driven by limited memory in embedded systems and by the importance of compiled applications transmitted over the Internet. The rise of embedded systems has also sparked interest in compiler techniques that can reduce the power consumed during execution [27, 28, 4, 15]. The classic approach to the appearance of a new objective function has been to formulate new transformations and add them to the optimizer.

The community has been building optimizing compilers for forty years. We know how to build optimizing compilers that produce efficient code for a single uniprocessor target. We can do this for most modern processors. We have learned to build compilers that are easier to retarget but produce less optimized code. Such "retargetable" compilers are used in many situations where the economics cannot justify a large standalone compiler effort. What we have not developed is an economical way to produce high-quality compilers for a wide variety of target machines.[1] Unfortunately, we have also learned that building high-quality compilers is expensive, primarily because it requires years of effort by experts, who are usually in short supply.

[1]The resources devoted to compiler development vary widely. Contrast, for example, the excellent code produced by the compiler for Cray's (Tera's) MTA architecture (with 1 or 2 installed systems) against the code quality achieved by typical compilers for Pentium-based systems (with a huge number of installed units).

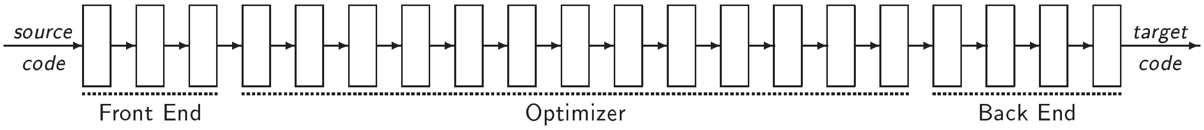

Figure 1: Classic Compiler Structure

This paper describes a framework for implementing optimizing compilers that will easily and automatically adapt their behavior to the circumstances under which they operate—to different applications, to different target-machine performance parameters, to different sets of transformations, and to different objective functions for optimization. Our goal is to change the economics of producing high-quality compilers in a fundamental way—by automating much of the work required to tune a compiler for new circumstances—and make it possible to build retargetable compilers that produce excellent code.

Today, we are exploring these ideas in a research compiler. We expect that, within five years, the combination of faster processors and better understanding will make adaptive compilation fast enough for routine use. These same improvements will then let us apply these ideas to harder problems in compilation, such as selecting the best strategy for data distribution and parallelization (using fast performance estimators).

2 A New Structure for Compilers

Since the 1960s, compilers have consisted of a fixed sequence of passes, applied in some preselected order, as shown in Figure 1. The compiler runs through the same sequence of passes on every input program. For example, the original Fortran translator had six passes [3, 2], the classic Fortran H compiler had ten [18, 23], and the Bliss-11 compiler had seven [31]. IBM's PL.8 compiler [1] and HP's compiler for the PA-RISC [14] also followed this same basic organization. Modern systems retain this basic structure, with more passes. For example, the SUIF compiler has eighteen or more passes for its optimizer [16], and the recently released Pro64 compiler from Silicon Graphics has over twenty passes [25]. These compilers differ in the number of passes, the selection of specific algorithms for these passes, and the order of application for those passes, but their basic structure remains the same.

The choice of specific transformations and an order for their application play a major role in determining the effectiveness of an optimizing compiler. We call an ordered list of transformations a *compilation sequence*. Since the 1960s, compiler writers have chosen compilation sequences in an *ad hoc* fashion, guided by experience and limited benchmarking. Efforts to find the "best" sequence have foundered due to the complexity of the problem. Transformations both create and suppress opportunities for other transformations. Different techniques for the same problem catch different subsets of the available opportunities. (For example, different ways of performing redundancy elimination miss different cases [6].) Finally, combinations of techniques can achieve the same result as some single techniques.

Unfortunately, the best compilation sequence depends on many factors, including: 1) the specific details of the code being compiled, 2) the pool of available transformations, 3) the target machine and its performance parameters (which vary from model to model), and 4) the particular aspect of the code that the user desires to improve (*speed, space, power, page faults, etc*). Classic compilers try to address the second and third factors through design-time decisions, but ignore the first and last. This makes it difficult to predict the impact that changes in the compilation sequence will have on the compiled code. Today, we lack the knowledge to analytically predict the results of a particular sequence in a particular set of circumstances; this prevents a purely analytical process from deriving good code sequences. In this paper, we describe a new approach

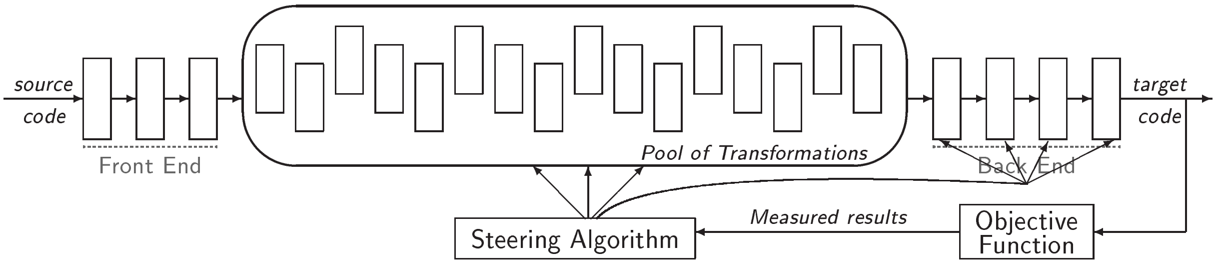

Figure 2: Structure of the new compiler

to structuring compilation that promises to simplify the construction of high-quality optimizing compilers across a wide variation in all four of these factors.

Our new system, shown in Figure 2, replaces the fixed-order optimizer with a pool of transformations, a steering algorithm, and an explicit, external objective function. The steering mechanism selects a compilation sequence and compiles the program with that sequence. The compiler evaluates the objective function on the resulting target-machine program. The measured results serve as input to the steering algorithm, allowing it to refine its choices and to explore the space of possible compilation sequences. Through repeated experiments, the steering algorithm discovers a compilation sequence that minimizes the objective function, given the source code, the available transformations, and the target machine.

This approach addresses one of the fundamental challenges in the design and implementation of an optimizing compiler—choosing a specific set of transformations and an order of application for them—by *computing* the solutions. It relies on the speed of modern computers to replace the fixed-order compiler of the 1960s with a structure that adapts to new performance parameters, new input programs, new transformations, and new objective functions. It applies inexpensive cycles to solve a problem in compiler design that has defied both theory and practice for forty years. It makes the compiler's objective function explicit, changeable, and multi-dimensional rather than implicit, fixed, and one-dimensional. The resulting compilers can optimize for a variety of objectives (including running time, code size, page faults, and energy consumption) and for combinations of those objectives.

We have done preliminary experiments using a particular search technique to find program-specific compilation sequences [9, 24]. To date, we have experimented with objective functions that optimize for code size (generating compact executables), that optimize for speed (generating fast code), and that optimize for a property related to power consumption (generating power-efficient code). In this paper, we describe the computation of the best compilation sequence as a sequential search problem. We then show that biased-random sampling of the combinatorial space of possible compilation sequences is an effective means of finding good solutions. The following sections describe our approach, related prior work, and our preliminary experimental results in more detail.

3 Searching for Compilation Sequences

Currently, most effective compilers include ten to twenty transformations, drawn from the hundreds that have been proposed in the literature. Picking the best compilation sequence for a specific program and a given objective function is hard: 1) there is little theoretical understanding of the effect of particular compilation sequences on the external objective function, and 2) the space of compilation sequences is too large for approaches relying on exhaustive search. Most compilers offer a small number of compilation sequences (`-O1`,

`-O2`, `-O3`, ...) discovered manually by designers. If none of these sequences is a good fit to the application or the user's real performance goals, the user has no recourse.[2]

Picking compilation sequences is an instance of a family of combinatorial problems called *sequential decision-making* problems. These problems have the following properties:

- Solving a problem requires making a sequence of decisions.
- The effect or outcome of each decision is a function of decisions made in the past as well as other random factors not entirely within the decision-maker's control.
- A decision made at a given point in time alters the set of choices for the future. At each step, the future impact of a decision must be considered.
- The objective function depends in a complex way on the interactions between the individual decisions and their stochastic outcomes.

The problem of finding compilation sequences for specific circumstances is such a problem. The traveling salesman problem (TSP) and other discrete combinatorial optimization problems are also members of this problem class. The standard approach to solving these problems uses deterministic or stochastic dynamic programming. Since traditional dynamic programming implementations need excessive amounts of space, complete search algorithms (*e.g.,* branch and bound algorithms for TSP) are used. Complete search algorithms guarantee a globally optimal solution. However, they are only practical for problems where effective pruning techniques are known. For example, the available theory on TSP permits the design of efficient heuristics (*e.g.,* the Lin-Kernighan heuristic) that prune unpromising paths early in the search. Unfortunately, too little is known about picking compilation sequences to enable this kind of early pruning. Without deeper analytical understanding, the search cannot differentiate between promising and unpromising subsequences.

Biased Random Sampling A key insight behind our approach is the use of randomization in the exploration of good compilation sequences. We design *adaptive sampling algorithms* that learn to construct good sequences by randomly sampling and evaluating them; by keeping statistics on how the various sequences perform with respect to the external objective function; and by using these statistics to guide future sampling. This is the paradigm of biased-random search or biased-random sampling. Initially, the algorithm performs a random walk in the space of all sequences, because, in the absence of any information, they all appear equally promising. As more sequences are sampled and evaluated, the search algorithm builds a probabilistic model relating subsequences to their evaluation, and biases the sampling strategy by these probabilities. Regions that yield unpromising results are sampled less often than regions that yield more promising results.

This adaptive, biased, randomized sampling strategy is the best that we can do without heuristics for pruning, and without (partial) analytical models of the interactions between sequence elements. The method performs better than pure random walk strategies. In our experiments that generated compilation sequences with a genetic algorithm (one kind of biased random sampler), we compared the rate of convergence of the genetic algorithm's search against that of an unbiased random walk strategy. The genetic algorithm found the "best" solution with far fewer probes than random sampling [9, § 4.3].

From a practical perspective, adaptive biased randomized sampling is an *anytime* algorithm. Because it retains the best result that it has seen, it can be stopped at any time. This lets us construct stopping criteria based on practical considerations such as resource limits and rate of change in solution quality. For example, a wall-time limited search will still return the best solution that it has discovered.

[2] Compilers used in benchmarking—an activity that directly affects sales of computers—often have myriad flags that let a benchmarking specialist hand-tune the compiler's behavior for a specific program. The effective use of these flags, individually and in combination, requires in-depth knowledge of both the compiler's inner workings and the application being compiled. This is precisely the sort of tuning that can and should be automated to make it routinely available to all users!

The general schema for adaptive, biased, randomized sampling includes a wide range of specific algorithms from applied mathematics and artificial intelligence. Examples include parallel direct search [10, 29], iterative repair [20, 26], algorithm selection by statistical sampling of search trajectories [17], and genetic algorithms [13]. These methods differ in how they use information from past searches to generate new candidate solutions, and how they choose starting points. Most of these techniques come with guarantees of convergence to local optima.

The Case for Biased Random Sampling In designing our prototype adaptive compiler, we chose to use biased-random sampling as a paradigm rather than pursuing an analytical technique to derive the appropriate transformation sequence.[3] If the compiler could predict, with reasonable accuracy, the impact of a specific compilation sequence on a specific program in a given execution context, it would open up other avenues of research for us to pursue.

Two factors make such predictions difficult and inaccurate today.

- The improvement from a given transformation, in isolation, varies widely as a function of detailed properties of the input program. An optimization targets specific opportunities; it can only improve the code if those opportunities exist. To estimate its impact with any accuracy, the estimator would need to discover those sites where the transformation applies and estimate each site's impact on execution time. Finding those sites is a large part of the transformation's work; in most cases, the cost of an accurate estimator would be roughly the same as the cost of performing the transformation. To complicate matters further, transformations sometimes produce negative results; for example, Briggs and Cooper reported improvements ranging from +49% to -12% for their algebraic reassociation technique [5].
- The interactions between transformations and their overlapping effects make isolated predictions inaccurate. Consider trying to predict the improvement that will accrue from performing transformation B in the compilation sequence ABC. The predictor could determine the number of occurrences of code sequences that B can improve and it can estimate how often they execute. However, A might rewrite the code to eliminate these opportunities, or it might move them to places where they execute less often. Similarly, B might eliminate opportunities for C or create situations where C's efforts are counter-productive. To perform accurate prediction, the predictor must operate on the code that is input to the transformation—the result of all previous transformations in the sequence. This fact, alone, makes accurate standalone prediction as expensive as actually compiling the code.

Until we understand much more about the behavior of optimizations in their compile-time context, analytical prediction of the behavior of sequences will remain impractical.

The principal argument against our sampling-based approach is its expense. Most of that cost accrues in the evaluation of proposed sequences. We are exploring techniques for approximations and proxies that estimate the behavior of a given sequence from the available evaluations of other related sequences. However, we recognize that the behavioral complexity of the optimizer may preclude the development of efficient approximators. Our approach is fast enough for a research compiler today. In five years, the combination of faster computers and better understanding of these adaptive compilers will allow routine use of such techniques in commercial compilers, and let researchers apply them to even harder problems in compilation such as strategies for data distribution and parallelization.

[3]The analytical approach has had limited success; for example, Lam *et al.* recast loop transformations to improve memory behavior into a framework of unimodular transformations and were able to analytically derive an appropriate sequence [30]. However, their model included a limited set of transformations that attacked a single problem—cache locality—with a single objective function. No generalization to include arbitrary optimization techniques has been proposed.

4 Related Prior Work

Previous attempts at building adaptive compilers [21] have focused on feeding dynamic profile information from program execution back into the compiler to guide optimization. Other attempts to use search in optimization include Nisbet's system, which used genetic algorithms in an attempt to parallelize loop nests [22, 11], and Massalin's Superoptimizer, which used exhaustive search in an attempt to perform optimal instruction selection [19]. Nisbet's system was ineffective, probably because search was not a good fit to his problem. Massalin's technique produced good results, but was too expensive for routine use. Granlund and Kenner adapted Massalin's ideas to produce a design-time tool that generates assembly sequences for use in GCC's code generator [12]. Our preliminary work using a genetic algorithm to find compilation sequences suggests that search is a good fit to the problem and that adaptive randomized sampling can yield good results in a reasonable amount of time.

5 Preliminary Experimental Results

We have built a preliminary system that uses biased random sampling to search for a compilation sequence that minimizes an external objective function. We have tested the system with two different sampling methods: genetic algorithms and hill climbing with randomized restarts. We have implemented three distinct objective functions:

1. *Code size:* This work was motivated by the need for compact code in embedded systems. Rather than compressing the code resulting from a standard compilation, we used a genetic algorithm to find compilation sequences that produced smaller code [9]. Search-based compilation resulted in code that was, on average, 13% smaller than the code produced by the compiler's original optimization sequence. Because we broke ties in favor of speed, the resulting code code was typically faster than the code produced by the original sequence.

   In contrast, adding direct compression (based on suffix trees and procedure abstraction) to the original compiler produced about a 5% reduction in code size [8]. Thus, search-based compilation shrank the code about 2.5 times more than the direct technique. Furthermore, procedure abstraction must slow down the code. In contrast,the compilation sequences found by the adaptive compiler almost always led to faster code.

2. *Running time:* As part of the same experiment [24], we reversed the order in which the objective function considered space and speed. When optimizing for speed, with space as its tie breaker, the prototype system produced code that was up to 20% faster and slightly smaller than that produced by the original compiler. Again, the combination of adaptive behavior and consistent tie-breaking led to sequences that improved both criteria.

3. *Inter-operation name transitions:* We have used the prototype system to investigate the impact that compilation can have on power consumption by the microprocessor. With an objective function that minimizes the number of inter-operation bit-transitions in the name fields of instructions[4], the prototype produced reductions of 20% from unoptimized code and 6 to 7% from the compiler's default compilation sequence—without adding new transformations to the compiler. To continue this work, we will add transformations to the compiler that enhance this effect; the adaptive compiler will discover when to apply them. This should increase the overall improvements in the objective function [7].

In each case, the adaptive compiler found a compilation sequence that produced better solutions than the original, fixed-sequence compiler. These sequences never included more passes than the original, fixed-sequence compiler. Most of the time, they contained fewer passes. The tie-breaking regimen in the first two

[4]Reducing the number of bits in the instruction stream that change from operation to operation leads to lower power consumption by the processor.

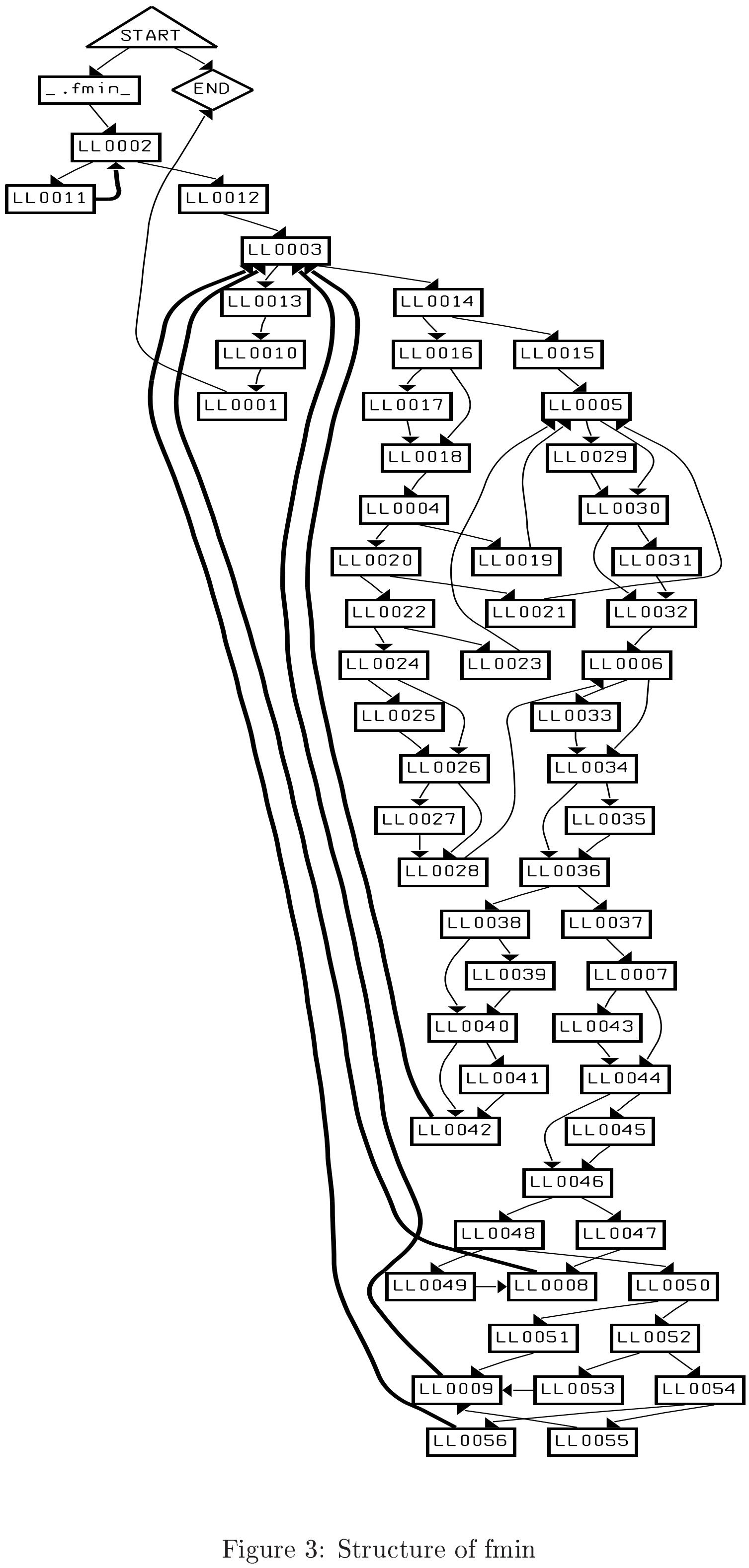

Figure 3: Structure of fmin

experiments actually created a two-dimensional objective function, with a clear priority that preferred one dimension over the other. We observed clear improvement in both dimensions.

Analysis of the Solution Space These preliminary experiments have shown the promise of adaptive compilation—finding program specific compilation sequences can produce code that more closely fits some stated objective. To understand whether or not our approach of biased random sampling is effective, we conducted another round of experiments. This work focused on a single benchmark program, `fmin`, that is small enough ($<$ 200 lines of Fortran) to permit near-complete mapping of the solution space, but complex enough to exhibit interesting behavior. Figure 3 shows the control-flow graph of its primary procedure. We optimized it against a simple objective function—the number of operations executed in a run of the program. Our goal was to address three questions:

1. What is the solution density of the space? That is, how many sequences of the M^n combinatorially possible ones (assuming M available transforms and sequences of length n) yield the lowest operation count? Are there many equally good solutions?
2. Are there local minima? How many of them are there, and how often will a random sampling strategy get stuck on them?
3. What is the distribution of solutions? Are they clustered tightly in space, or are they distributed all over? That is, what is the probability that a random sampling strategy will stumble into a good solution quickly?

Answers to these questions should dictate our approach to finding good program-specific compilation sequences. If good sequences are widely distributed and local minima are rare, then a good local search technique should, with high probability, find an optimal sequence. On the other hand, if good sequences are rare and the local minima are common, then local search will require restarts and the ability to move uphill from a local minimum. Finally, if the solution clusters are dispersed widely in the space, search techniques that explore multiple starting points in parallel and that mutate sequence guesses drastically in each iteration (such as genetic algorithms) become important.

While we recognize that the answers to these questions may be program specific, we began this work by focusing on a single program, `fmin`.[5] We used two randomized sampling strategies: hill climbing with random restarts and genetic algorithms.

The hill climber begins with a random sequence and mutates it, a single position at a time. If the change yields an improvement in the objective function, it is accepted. When the hill climber finds a sequence that it cannot improve, indicating a minimum (local or global), it records the sequence and its fitness value, and starts over by generating a new random sequence.

The genetic algorithm maintains a population of sequences. It tests each sequence by compiling the code and measuring the objective function. It creates the next generation of sequences using fitness-biased selection and crossover.[6]

The tables at the top of Figure 4 show two particularly good runs of the hill climber. The first column shows the ordinal number of the trial at which the sequence was derived. The second column shows the actual sequence; each letter denotes a specific optimization pass in the compiler. The third column shows the number of cycles required to run the resulting code on a simulated, single-issue Risc machine. In these two runs, the hill climber quickly got within 15% of the optimal solution (822 cycles). The first run required

[5] Larger experiments are running as this paper is being written and reviewed. At the symposium, we will have additional data.

[6] We have experimented with a number of strategies for crossover, for mutation, and for fitness scaling. Changing these parameters of the genetic algorithm do change its behavior, but do not affect the comparison with hill climbing.

Trial	Sequence	Cycles
0	nbavacaabrgozvv	1672
15	nbavacaabrgczvv	1360
30	nbovacaabrgczvv	1204
45	nbovacaabrgczpv	1196
60	nbovacaabrvczpv	1192
150	nbovacaabdvczpv	1103
165	nbovacaabdvczps	954
180	npovacaabdvczps	940
195	npovlcaabdvczps	939
210	npovncaabdvczps	937
255	npovncaabdvpzps	936
270	npopncaabdvpzps	933
285	npopncaabdnpzps	931
315	npopzcaabdnpzps	913
345	npopzcaapdnpzps	910
360	npopzcaspdnpzps	841
375	npopzcaspdnpzgs	838
390	ppopzcaspdnpzgs	834
435	ppopzcaspdnpzbs	833
525	ppopzpaspdnpzbs	831
750	ppoplpaspdnpzbs	830

Trial	Sequence	Cycles
0	togmolrdncqdatc	2583
15	togmolpdncqdatc	1118
30	togmolpancqdatc	1113
45	togmolpancqdasc	964
60	togmolpancqdbsc	951
75	togmolpsncqdbsc	876
90	togmolpsncqdnsc	860
150	togpplpsncqdnsc	842
285	togpblpsncqdnsc	840
359	togpblpsncpdnsc	839
374	pogpblpsncpdnsc	838
449	pogpblpspcpdnsc	837
2142	poppblpspcpdnsc	834
2232	poppbltspcpdnsc	833

Sequences from two hill climber runs

a	Assertion insertion
b	SCC-based value numbering
c	Global constant propagation
d	Dead code elimination
g	Partition-based value numbering
l	Partial redundancy elimination
m	Global renaming
n	Useless control-flow elimination

o	Logical peephole optimization
p	Loop peeling
r	Reassociation
s	Copy coalescing
t	Strength reduction
v	Local value numbering
z	Lazy code motion

Interpreting the sequences

Figure 4: Hill climber runs

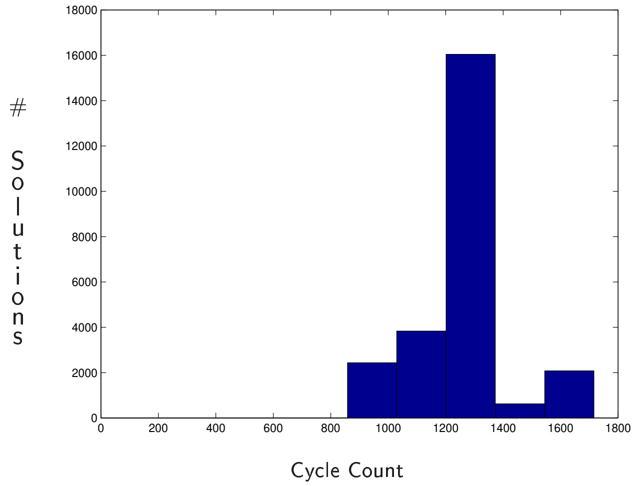

Figure 5: Distribution of solutions for `fmin`

180 trials to get below 948 cycles, while the second required 75 trials. (Each trial mutates one position and evaluates it.) However, further improvement takes much longer.

Neither run converges to the minimum cycle count. These sequences, along with other runs that are not shown, demonstrate that the solution space contains a significant number of local minima, and that those minima are dispersed through the solution space. This suggests that parallel exploration of multiple sequences and mechanisms for generating new sequences that are more powerful than single-position mutation will be necessary to obtain faster convergence.

The hill climber found solutions that were within a small percentage of the best solution. However, its progress slowed markedly as the objective function values approached the minimum. A genetic algorithm, with variable length sequences and a strategy that eliminates duplicates by mutating them until they are unique, outperforms the hill climber. The best sequences found by the genetic algorithm are:

Sequence	Cycles
odspspplsn	822
poplppcppvbdpsn	825

Note that it will take several single-position mutations to transform the sequences found by the hill climber into the second sequence found by the genetic algorithm. Furthermore, the sequence found by the hill climber is a local minimum. This suggests that the hill climber must cross many deep valleys before it finds the deepest valley in the space. To find the true minimizer in this complex space will require a search algorithm capable of leaping over those valleys.

To better visualize the solution space, we chose five of the mostly commonly occurring transformations in the sequences for `fmin` found by the genetic algorithm. Using a sequence of length 10, we ran each of the 5^{10} sequences through the optimizer to determine how many of those sequences yield the smallest cycle count. The histogram in Figure 5, based on 1% of the overall sequences, shows the distribution of cycle counts for sequences drawn from the 5^{10} possible sequences. They show that the good solutions ($<$ 830 cycles) are extremely rare, about 1 in 30,000. Our experience with the hill climber suggests a preponderance of local minima, testifying to the complexity of the solution space. These results also suggests the possibility that there may not be an algorithmic solution to finding the best sequence.

6 The Promise of Search-based Compilation

Our long-term goal for this work is to address the economic problem that confronts compiler-writers: how to handle the rapid proliferation of processors, applications, objective functions, and environmental constraints without abandoning high-quality optimization. It is not economically feasible to produce distinct compilers for all of these circumstances using current practices. Without a new way to organize, build, and tune optimizing compilers, we will be forced to accept poor quality code that fails to meet the users' real needs.

Adaptive compilers based on search open up new vistas for research and for practical application. They create a market economy for transformations, allowing competitive evaluation of different techniques on a level playing field. They create an environment where the compiler writer can include niche transformations—those with high payoff but narrow applicability. They let us apply the power of search to new issues in compilation. For example, to discover an appropriate compilation sequence for a Java JIT, the compiler writer might limit the sequence to three transformations and optimize for run-time speed plus compile time.

Our strategy—building adaptive compilers and the tools to automate the process of configuring them—ties the strength of our compilers to the speed of our machines. This is a resource that compilers have not exploited well in the past. Adaptive compilers will broaden the range of input programs that routinely attain good performance. They will be responsive to new performance goals, expressed in the form of new external objective functions. They will easily accommodate new results from the research of others, since new transformations can simply be added to the pool.

7 Remaining Challenges

Building a robust adaptive optimizing compiler will be a major challenge. It will require strategies to minimize explicit, user-selected objective functions over a complex and poorly characterized space—*viz.*, the combinatorial set of distinct compilation sequences. It will require new techniques for implementing optimizations in a modular fashion, managing their reconfiguration, and measuring the results of each compilation. It will require careful consideration of stopping criteria, of strategies for producing and storing results incrementally, and of how to apply the knowledge gained by exploring the space of compilation sequences. It will require a major implementation effort, without which we cannot evaluate the effectiveness of our ideas.

7.1 Engineering for Reconfiguration

The diagram in Figure 1 makes it appear that classic compilers are modular and that their components can be reordered. In practice, inspecting the internals of a compiler usually reveals critical ordering constraints imposed by the implementation. These constraints may be explicit, such as when one pass allocates a structure and fills it with information, and a later pass uses that information and frees the structure. Constraints may also be implicit, with one pass relying on others. This approach, which can simplify the implementation of some passes by keeping them narrowly focused, was popularized by IBM's PL.8 compiler [1]. Even in compilers designed with modularity as a goal, subtle order-dependences arise from these implicit constraints.[7]

These inter-pass constraints, both explicit and implicit, are a major impediment to the construction of adaptive compilers. The heart of our system is a pool of transformations that can be run in, essentially, arbitrary order. We envision two distinct but complementary efforts to solving the problem of engineering reconfigurable compilers: one aimed at a set of design principles for writing new transformations, and another

[7] Our early experiments with genetic algorithms to find compilation sequences in the MSCP framework exposed several order-dependences that had gone undetected in many years of use.

aimed at understanding the constraints that exist and ensuring that proposed compilation sequences do not violate those constraints.

7.2 Deriving Practical Compilers

The adaptive compilers that result from this work will allow researchers and compiler writers to explore the space of compilation sequences and the impact of those sequences on code quality. To make these ideas useful in practice, we must design mechanisms that use the results of full-blown, adaptive compilations in limited-time compiles. To build efficient production compilers from this configurable base will require additional research and implementation. We intend to explore several alternative strategies.

- Training the compiler on a representative set of programs to find the k best sequences, and having the production compiler try all k sequences on an input program to find the best result. For small k, say 3 to 5, this should produce some of the benefit while limiting the increase in compile time.
- Adopting an incremental strategy for sampling, where the compiler retains partial results across compiles. This trades a minor cost increase on each compile for a long-term improvement in code quality. This scenario may require a periodic, randomized disturbance of its model to help it avoid local minima based on historical evolution.
- Invoking the full algorithm, with a strict wall-time limit, so that the compiler returns the best code it can find in an hour, or five hours. This provides a direct way of limiting resource use; it might work well coupled with an incremental sampling strategy (above).
- Limiting the focus of the adaptive compiler to the performance critical routines in an application. This would require a mechanism for identifying those routines—both user directives and run-time profiling merit investigation.

By varying the size of the training set, the maximum length of the compilation sequence, and the amount of time allowed for the search, the compiler writer should be able to achieve a variety of effects. For particularly important codes, the user may want a version that limits its training set to that program.

As we gain experience with adaptive, sampling-based compilation, we hope to learn enough about the behavior of the optimizations and their interactions to allow the compiler to perform all or part of the search analytically. Strategies that might fit this model include building a database of program characteristics versus compilation sequences; restricting the set of optimizations to a smaller set that has predictable effects and interactions; and using wholesale approximations of the effects. We hope to learn enough to let the compiler prune the search space aggressively and early, eliminating unproductive search paths. This may lead to techniques that make complete search, with pruning, tractable.

Acknowledgements

Many people have contributed to our understanding of these issues, to our initial experiments in this arena, and to our ability to describe our vision for adaptive compilation. The members of the scalar compiler group at Rice, past and present, have built tools that let us explore these ideas. Phil Schielke, Tim Harvey, Steve Reeves, and L. Alamagor have all contributed with insight, hard work, and long discussion. Stephanie Forrest at the University of New Mexico suggested a number of improvements to our initial genetic algorithm. Our initial work on adaptive compilation to reduce code size was supported by DARPA through USAFRL contract F30602-97-2-298. Our work on reducing power consumption is supported, in part, by the State of Texas through its Advanced Technology Project. The remainder of this work has been supported by the Los Alamos Computer Science Institute.

References

[1] Marc A. Auslander and Martin E. Hopkins. An overview of the PL.8 compiler. *SIGPLAN Notices*, 17(6):22–31, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*.

[2] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The FORTRAN automatic coding system. In *Proceedings of the Western Joint Computer Conference*, pages 188–198, February 1957.

[3] John Backus. The history of Fortran I, II, and III. In Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981.

[4] Jonathan G. Bradley and Gene A. Frantz. DSP microprocessor power management: A novel approach. Texas Instruments Technical White Paper, 1998.

[5] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. *SIGPLAN Notices*, 29(6):159–170, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[6] Keith D. Cooper. Why is redundancy elimination hard? Excerpt from talk at Rice Computer Science Affiliates Meeting. Available at `http://www.cs.rice.edu/~keith/1960s`, October 2000.

[7] Keith D. Cooper and Tim Harvey. A study of estimated name transitions in Fortran codes. Technical report in preparation, available on the web at http://softlib.rice.edu/MSCP/Publications.html, April 2001.

[8] Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded RISC processors. In *Proceedings of the ACM SIGPLAN 99 Conference on Language Design and Implementation*, May 1999.

[9] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, May 1999.

[10] John E. Dennis and Virginia Torczon. Direct search methods on parallel machines. *SIAM Journal on Optimization*, 1(4):448–474, November 1991.

[11] Nicolas G. Fournier. Enhancement of an evolutionary optimising compiler. Master's thesis, Department of Computer Science, University of Manchester, September 1999.

[12] Torbjörn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. *SIGPLAN Notices*, 27(7):341–352, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*.

[13] J.H. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[14] Mark Scott Johnson and Terrence C. Miller. Effectiveness of a machine-level global optimizer. *SIGPLAN Notices*, 21(7):99–108, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*.

[15] M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Proceedings of the International Symposium on Computer Architecture*, June 2000.

[16] Monica Lam and the SUIF group. Documentation with the SUIF-2 compiler release. *Available from the* SUIF *web site,* `http://suif.cs.stanford.edu` .

[17] L. Lionel and M. Lematre. Branch and bound algorithm selection by performance prediction. In *AAAI-1998*, 1998.

[18] E.S. Lowry and C.W. Medlock. Object code optimization. *Communications of the ACM*, pages 13–22, January 1969.

[19] Henry Massalin. Superoptimizer – A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–126, Palo Alto, CA., 1987.

[20] S. Minton, M.D. Johnston, A.B. Phillips, and P. Laird. Minimizing conflicts: A heuristic method for constraint-satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[21] Thomas Night. `http://www.ai.mit.edu/projects/transit/tn101/tn101.html`. Web site for the Transit Project.

[22] Andy Nisbet. GAPS: Iterative feedback directed parallelisation using genetic algorithms. In *Proceedings of the Workshop on Profile and Feedback Directed Compilation*, Paris, FR, June 1998. Workshop held in conjunction with PACT '98.

[23] Randolph G. Scarborough and Harwood G. Kolsky. Improved optimization of fortran object programs. *IBM Journal of Research and Development*, 24(6):660–676, November 1980.

[24] Philip J. Schielke. *Stochastic Instruction Scheduling.* PhD thesis, Rice University, Department of Computer Science, May 2000.

[25] Silicon Graphics, Inc. Documentation with the SGI PRO64 compiler release. SGI *released the compiler in open-source form during the summer of 2000. Code is available from the company.*, 2000.

[26] J. Thornton and A. Sattar. Using arc weights to improve iterative repair. In *AAAI-1998*, 1998.

[27] V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy: An overview. In *Proceedings of the 1994 IEEE Symposium on Low Power Electronics*, 1994.

[28] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on VLSI Systems*, 2(4), 1994.

[29] Virginia Torczon. Direct search methods for unconstrained optimization on either parallel or sequential machines. Technical Report 92-09, Rice University, Department of Computational and Applied Mathematics, 1992.

[30] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[31] William Wulf, Richard K. Johnson, Charles B. Weinstock, Steven O. Hobbs, and Charles M. Geschke. *The Design of an Optimizing Compiler.* Programming Language Series. American Elsevier Publishing Company, 1975.